Integrated Development System

Dynamic C™

Introduction

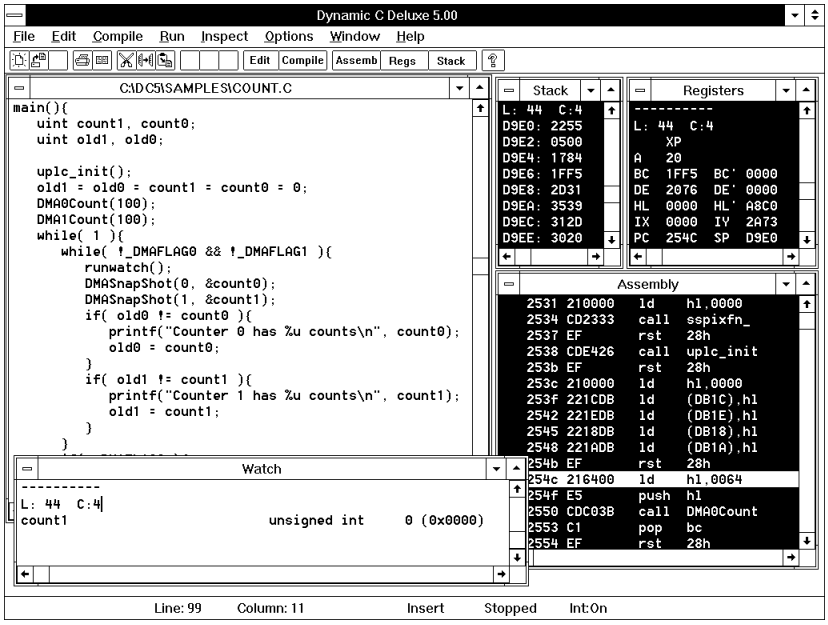
Dynamic C™ is the development system by which you develop software that runs on your Z-World controller. Dynamic C incorporates an easy-to-use program editor, fast C compiler, and source-level debugger. The C compiler has enhancements that support embedded systems development.

Dynamic C runs in Windows (3.1, NT, or 95) on your IBM-compatible PC and is designed specifically for Z-World controllers and control products.

There are two versions of Dynamic C:

Standard	Limited to 80K bytes of machine code.
Deluxe	Not limited and fully supports extended memory.

At this writing, Dynamic C is at revision 5.2.



Why C?

Using a programmable controller is the most flexible way to develop a control system. C is the preferred language for embedded systems programming. It is widely known and produces efficient and compact code. Because it is a high-level language, you can develop code much faster than you could with assembly language, or some of the machine languages offered by PLC manufacturers. Yet, C allows you to program at the machine level whenever you want.

C is suitable for complex applications as well as simple ones. C has floating point math with a substantial mathematical function library. C allows you to develop complex control algorithms when you need to do so.

The Nature of Dynamic C

Dynamic C integrates the following development functions

Editing, Compiling, Linking, Loading, Debugging

into one program. In fact, compiling, linking and loading are one function: Dynamic C compiles code directly to your target controller (or to a file).

Dynamic C has an easy-to-use built-in text editor. Most Windows users will immediately know what to do.

With the symbolic debugger, you can execute and debug programs interactively at the source-code level.

Ultimately, you'll create EPROM files or down-loadable files for programs that run stand-alone in the controller.

Because all the functions are integrated, (1) you can switch from one function to another with a simple keystroke, (2) the debugger has access to all the compiler information, and (3) you can monitor your controller directly.

Dynamic C also supports assembly language programming. You do not have to leave C or the development system to write assembly language code. You can mix C and assembly language, line by line, in your program.

Dynamic C has the following debugging windows:

STDIO	("standard I/O") allows the program running in your controller to print messages on your development screen.
Assembly	Displays an assembly view of compiled code.
Watch	Allows you to type and evaluate expressions, monitor or set variables, and call functions at will.
Register	Displays register contents and status bits.
Stack	Displays the top 8 bytes of the processor stack.

Dynamic C's debugger allows you to set and clear breakpoints on-the-fly, to single-step with and without descent into functions, and to view execution at the assembly level as well as at the source-code level.

Dynamic C provides extensions to the C language (such as **shared** and **protected** variables) that support real-world system development. You can write interrupt service routines in C. Dynamic C supports real-time multi-tasking with its real-time kernel and its **costatement** extension.

Dynamic C comes with many function libraries, all in source code. These libraries support real-time programming, machine level I/O, and provide standard string and math functions.

References

Please refer to

- *The C Programming Language* by Kernighan and Ritchie, published by Prentice-Hall.
- *C: A Reference Manual* by Harbison and Steel, also published by Prentice-Hall.
- *Z180 MPU User's Manual*
- *Z180 Serial Communication Controllers*
- *Z80 Microprocessor Family User's Manual*
- *Microsoft Windows User's Manual*.

Contents

Dynamic C compiles, links and loads in one pass—directly to your target. On a fast PC, Dynamic C can compile more than 250 lines of source code per second. Thus, a large program—say 8,000 lines of code—might generate 80K bytes of machine code and take about 30 seconds to compile.

Dynamic C is Different

In an embedded system, there is no operating system or supervisor that can halt a program if it goes wrong or perform services for the program. An embedded program has to do it all, and handle its own errors and keep on running. An embedded program also has to initialize itself.

In an embedded system, a program usually runs from EPROM (or flash) and uses a separate RAM for data storage.

Often, an embedded system comprises a number of concurrently executing tasks, rather than a single task.

Dynamic C specifically supports embedded systems.

Differences from ANSI C are summarized here and discussed following this summary:

- The default storage class is **static**, not **auto**.
- There is no **#include** directive, nor are there any include (header) files. There is a **#use** directive.
- Variables that are initialized when declared are considered *named constants* and placed in ROM. It is an error to try to change such "variables."
- Dynamic C has a unique concept: *function chaining*.
- "Costatements" allow multiple concurrent tasks in a single program.
- You can write interrupt service routines in C.
- Dynamic C has **shared** and **protected** keywords that help protect your data from unexpected loss.
- Dynamic C has a set of features that allow you to make fullest use of extended memory.
- The **extern** keyword has altered meaning. The **register** keyword has altered meaning.
- Dynamic C has a **subfunction** extension that lets you optimize frequently used code.
- Dynamic C does not support enumerated types.
- Dynamic C allows embedded assembly code.

Default Storage Class

Unlike traditional C compilers, the default storage class for local variables is **static**, not **auto**.

Although this fact is disconcerting to many programmers at first, **static** storage is preferrable in embedded systems.

Initialized Variables

Static variables initialized when they are declared are considered *named constants*. The compiler places them in the same area of memory as program code: in EPROM or in flash memory. Uninitialized variables are placed in RAM, and are initialized by your application program.

Function Chaining

Function chaining, a concept unique to Dynamic C, allows you to distribute special segments of code in one or more functions. When a named function chain executes, all the segments belonging to that chain execute. Function chains allow your software to "switch modes" momentarily to perform global initialization, data recovery, or other kinds of tasks, at your request.

Dynamic C has two directives, **#makechain** and **#funcchain**, and one keyword, **segchain**.

- **#makechain** *chain_name*

  Create a function chain. When your program executes the named function chain, all of the functions or chain segments belonging to that chain execute.

- **#funcchain** *chain_name name*

  Add a function (or another function chain) to a function chain.

- **segchain** *chain_name* **{** *statements* **}**

  Define a program segment (enclosed in curly braces) and attach it to the named function chain. Function chain segments defined with **segchain** appear directly after data declarations and before executable statements:

```
my_function(){
    data declarations
    segchain chain_x{
        some statements which execute under chain_x
    }
    segchain chain_y{
        some statements which execute under chain_y
    }
    function body which executes when my_function is called
}
```

Your program will call a function chain as it would an ordinary void function that has no parameters. For example, if your function chain is named **recover**, this is how to call it:

```
#makechain recover
...
recover();
```

Dynamic C software comes with several built-in function chains, including **_GLOBAL_INIT** described next.

Global Initialization

Embedded systems typically have no operating system to perform services such as initialization of data. Further, various hardware devices in a system need to be initialized not only by setting variables and control registers, but often by complex initialization procedures. For this purpose, Dynamic C has a specific function chain: **_GLOBAL_INIT**.

You can perform any global initialization you want by adding segments to the **_GLOBAL_INIT** function chain, as shown under *Function Chaining*, above.

Have your program call **_GLOBAL_INIT** during program startup, or upon hardware reset. This function chain executes all **_GLOBAL_INIT** segments in your program (and in Dynamic C libraries as well).

Costatements

Dynamic C provides a capability whereby your program can execute a set of tasks concurrently. A data structure, some additions to the C language, and some functions comprise what Z-World calls *costatements*. A costatement is a construct—a block of code—that can suspend its own execution, thereby allowing other code to execute. A *set* of costatements execute, presumably, in an endless loop. All of the tasks in the set are in states of partial completion.

For further detail, refer to the section *Costatements*, later in this document.

Interrupt Service Routines

You can write interrupt service routines in C. The keyword **interrupt** designates an interrupt service routine:

```
interrupt my_handler(){
    ...
}
```

Shared and Protected Variables

You can declare variables **protected**. If your system resets while a protected variable is being modified, the variable's value can be restored when the system restarts.

A system that shares data among different tasks or among interrupt routines can find its shared data corrupted if an interrupt occurs in the middle of a write to a multibyte variable (such as type **int** or **float**). Declaring a multibyte variable **shared** ensures that any change to the variable is a *complete* change. (Interrupts are disabled while the variable is being changed.)

Extended Memory

Dynamic C supports the 1-megabyte physical address space of the Z180 microprocessor. This is called ***extended memory*** since the Z180 logical address space is 64K (16-bit addresses). Under normal circumstances, Dynamic C takes care of memory management for you.

Dynamic C also has keywords (such as **xdata** and **xstring**), functions (such as **xgetstring**), and directives (such as **#memmap**) that help you manage code and data in the extended memory space. See *Physical Memory*.

External Functions and Data

The keyword **static** cannot apply to functions. The meaning of the keyword **extern** is this:

- A variable or function is declared **extern** if it is defined in your target controller's BIOS.
- Declare a variable **extern** if it is to be defined later in the program or in another file.

Dynamic C has no **#include** directive, but does have a **#use** directive. Z-World's **#use** directive identifies a library from which functions and data may be taken. The file LIB.DIR contains the names of all known libraries. The file DEFAULT.H contains several sets of **#use** directives, one set for each controller Z-World offers. You may modify either of these files.

Dynamic C functions are not compiled separately and then linked. There are no precompiled software libraries. Dynamic C uses *source-code* libraries, from which necessary functions are extracted during compilation.

Dynamic C libraries make global variables and function prototypes available with special headers like this one:

```
/*** BeginHeader my_proc, my_func, my_var */
     void my_proc( int j );
     float my_func( float arg );
     extern int my_var;
/*** EndHeader */
```

If you create libraries, you must (1) create such headers to make your functions known to the Dynamic C compiler and (2) add the name of your library to LIB.DIR.

Subfunctions

Subfunctions allow often-used code sequences to be turned into a local "subroutine" within a C function. For more detail, see *Efficiency* later in this document.

Z-World Controllers and Dynamic C

Z-World controllers are based on the Z180 microprocessor. The Z180 is a well-established and popular microprocessor. A descendent of the original Z80 microprocessor, the Z180 also has the following on-chip subsystems:

- Dual 16-bit programmable timers
- Dual asynchronous serial communication ports
- A clocked serial communication port
- Dual DMA channels for high-speed data transfer between memory and I/O devices.

Many Z-World controllers use a Zilog PIO, SCC or KIO chip for additional I/O capability.

Physical Memory

Although Z-World controllers can address up to 512K bytes of ROM (or 256K flash), and 512K bytes of RAM, it is often not necessary to have memory chips this large on controllers. Typical memory chips have 32K or 128K bytes.

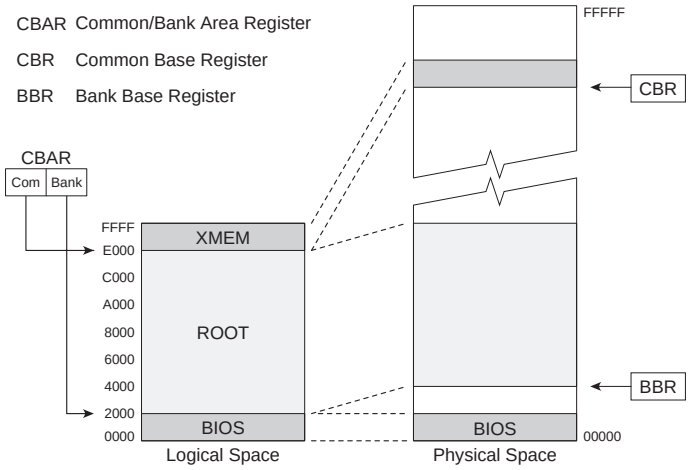
Code and constants are placed in ROM (or flash). Variable data (including the system stack) are placed in RAM.

ROM addresses start at 0. RAM always starts at a fixed address (usually 512K, or 80000_H).

Memory Management

Z180 instructions can specify 16-bit addresses, giving a *logical* address space of 64K (65,536) bytes. Dynamic C supports a 1-megabyte *physical* address space (20-bit addresses). An on-chip memory management unit (MMU) translates 16-bit Z180 addresses to 20-bit memory addresses. Three MMU registers (CBAR, CBR, and BBR) divide the logical space into three sections and map each section onto physical memory.

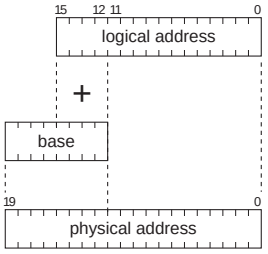
The following diagram illustrates the memory management registers and address mapping.



The logical address space is partitioned on 4 kbyte boundaries. The upper half of the CBAR identifies the boundary between **ROOT** memory and **XMEM**. The lower half of CBAR identifies the boundary between the BIOS and ROOT. The start of the BIOS is always address 0. The two base registers CBR and BBR map XMEM and ROOT, respectively, onto physical memory.

XMEM is a small window into physical memory. Its mapping will vary. The ROOT is a larger window into physical memory, but its mapping does not vary. The BIOS mapping is always fixed at address 0.)

Given a 16-bit address, the Z180 determines, using the CBAR, whether the address is in XMEM, BIOS, or ROOT. If the address is in XMEM, the Z180 uses the CBR as the base to calculate the physical address. If the address is in ROOT, the Z180 uses the BBR. If the address is in the BIOS, the Z180 uses a base of 0.



A physical address is, essentially,

(base << 12) + logical address

The Memory Partitions

The meaning of the partitions is this:

Name	Size	Description
BIOS	8K	Basic Input/Output System, containing the power-up code, the communication kernel, and important system features.
ROOT	48K	The area between the BIOS and XMEM (the bank area). The root—"normal" memory—resides in a fixed portion of physical memory. Root *code* grows upward in logical space from address 2000 (hex) and root *data* grow down from E000.
XMEM	8K	XMEM is an 8K "window" into extended physical memory. XMEM can map to any part of physical memory simply by changing the CBR.

Functions may be classified as to where Dynamic C is allowed to load them:

Key	Description
root	The function is to be placed in root memory. It can call functions residing in extended memory.
xmem	The function is to be placed in extended memory.
anymem	This keyword lets the compiler decide where to place the function. A function's placement depends on the amount of reserve memory available.

Dynamic C memory management is automatic. You can control how Dynamic C allocates and maps memory with the commands of OPTIONS menu.

Watchdog Timer

Most Z-World controllers have a "watchdog" timer that will initiate a hardware reset unless your program signals the timer periodically (about once a second). A failed program will generally fail to "hit" the watchdog timer. The watchdog timer provides a natural way to perform fatal error recovery.

Real-Time and Multi-Tasking Operations

Dynamic C includes two real-time function libraries to support real-time multi-tasking operations. The *costatement* extension supports cooperative multi-tasking.

Restart (Reset) Conditions

Z-World embedded applications need to differentiate the causes of reset. Hardware resets are listed as follows:

Regular reset	The system /RESET line is pulled low and released.
Power failure reset	Power drops below a threshold, and the supervisor chip pulls /RESET low and causes a reset.
Watchdog reset	The watchdog timer was not resest. It pulls /RESET low and causes a reset.

In addition to these hardware resets, an application may cause a *super reset*. Z-World's super reset is a mechanism to initialize certain persistent data in battery-backed RAM. A normal reset does not initialize these data, but *retains* their values. A super reset always occurs when a program is first loaded. Subsequent resets are normal resets, unless your software performs a super reset intentionally.

Reset Differentiation

Dynamic C include a set of functions to differentiate the various resets. These functions are grouped into two categories.

1 The function names begin with underbar (_), have important side effects, and may only be called *once and only once* at the beginning of your **main** program.

```
int _sysIsSuperReset()
int _sysIsPwrFail()
int _sysIsWDTO()
```

2 The function names do not begin with underbar, have no side effects, and may be called anywhere in your program.

```
int sysIsSuperReset()
int sysIsPwrFail()
int sysIsWDTO()
```

The **_sysIsSuperReset** function returns 1 if a super reset was requested and 0 if not. If a super reset was requested, this function calls **_prot_init** which initializes the protected variable feature. In addition, it calls the function chain **sysSupRstChain**. You may add any code you like to this function chain. If a super reset was not requested, this function calls **_prot_recover** which *recovers* partially written protected variables (if there are any).

The **_sysIsPwrFail** function returns 1 if a power failure occurred and 0 otherwise. You cannot use a custom power-failure handler with this function.

The **_sysIsWDTO** function returns 1 of a watchdog timeout occurred and 0 otherwise.

Reset Generation

Your software can generate two types of system reset. The function **sysForceReset** causes a watchdog reset. The function **sysForceSupRst** causes a super reset.

Instruction Timing

The Z180 has a relatively efficient instruction set. At 9.216 MHz, many instructions take about 1 microsecond. Floating point arithmetic is accomplished in software. Refer to the table following. Times are given in microseconds.

Arithmetic

Times required to perform basic mathematical operations.

Operation	6.144	9.216	12.288	18.432
16 bit integer add	4.2	2.8	2.1	1.4
16 bit integer multiply	14.7	9.8	7.4	4.7
16 bit integer divide	142.2	94.8	71.1	47.4
Long (32-bit) integer add	31.7	21.1	15.8	10.6
Long integer multiply	129.9	86.6	65.0	43.3
Long integer divide	605.3	403.5	302.6	201.8
Floating point (32-bit) add or subtract	104.4	69.6	52.2	34.8
Floating point multiply	174	116.0	87.0	58.0
Floating point divide	416.6	277.7	208.3	138.9
Sine or cosine	4672.8	3115.2	2336.4	1557.6
Square root	1277.3	851.5	638.6	425.8

Logical Decision Making

Execution times of various logical and counting operations are shown in the list below. Times are given in microseconds for a 9.216 MHz clock.

`if( k )...`	2.6 µs
`for(k=0;k<100;k++){...}`	loop overhead 12.8 µs
`func(n);`	call overhead with 1 integer argument 5.8 µs
`switch(n){...}`	22 µs for first case, 7 µs for each additional
`costate{...}`	costatement entry and exit overhead 32 µs
`waitfor(...)`	19 µs

Other Operations

Interrupt latency (guaranteed response to an interrupt) less than 100 µs. Sustained data throughput, interrupt driven at least 20,000 bytes per second. Burst data transfer rate using DMA at least 500,000 bytes per second.

Usage

Writing Programs

You'll be using one or more Dynamic C text windows to enter program text. You'll use the same editing techniques as you would in other Windows applications.

You'll create (1) programs and/or (2) function libraries.

Compiling Programs

Dynamic C gives you several ways to compile programs:

Compile to Target

Dynamic C compiles directly to your target controller. If your controller has flash memory, Dynamic C places code in flash memory. If your controller has EPROM, Dynamic C places code in RAM. Dynamic C communicates with your controller through a PC serial port. If your compilation is successful, Dynamic C enters *run mode* and maintains communication with your target.

Compile to File

Dynamic C compiles your program to a file whose nature and format can be selected with compiler options. Compile-to-File takes target information from your controller you have connected to your PC.

Compile to File with RTI File

Dynamic C compiles a program to a file *without* having a target controller present. If you wish to compile programs in this way, you must first create a *Remote Target Information* (.RTI) file for the specific controller in which the program will run.

Compiler Options

Code with BIOS	Generates an EPROM file containing your program and the BIOS of the target controller. Optionally generates an Intel hex format file in addition.
Null device	Generates no output. Useful if you just want to (1) perform syntax checking or type checking or (2) get the sizes of each code and data segment.
DLP for download	Generates a downloadable program file to be used by the Z-World download manager. Refer to the section *Remote Download* for detail.

Debugging Programs

Once you sucessfully compile a program to a connected target controller, Dynamic C enters *run mode*, or *debug mode*. There are two general debugging methods.

1 Make calls to **printf**. Dynamic C has an option to save all contents printed to the STDIO window in a file.

2 Probe and test the program, as it runs. Use breakpoints, single-stepping, watch expressions and the various debug windows. Dynamic C provides a variety of windows to monitor your program's state:

watch window	examine variables, evaluate expressions, and call functions
STDIO window	calls to **printf** use the STDIO window
assembly window	examine, or step, (dis)assembled code
register window	the Z180 register values, past and present
stack window	shows the top of the processor stack.

You can scroll the assembly, register, stack, STDIO, and watch windows to view the history of your program.

The Dynamic C debugger is symbolic. (1) Your executing program is linked to source code. The part of your program that is executing is highlighted in the source code window. (2) When you evaluate expressions, variables, and functions, it is in C language, using the names in your application, and normal integer, floating, and character representations of constants. (You can view your program at the machine level if you want to.)

Single Stepping

There are two commands for ***single stepping:***

- Trace into allows descent into function calls
- Step over prevents descent into function calls

When you issue one of the single-stepping commands, the current statement executes, debugging windows are updated, and the execution cursor is advanced to the next statement in the execution sequence.

Breakpoints

At times, you will want to run your program at full speed and then stop at ***breakpoints***. You can place them (and remove them) anywhere in your source code, at run-time.

There are ***hard*** and ***soft*** breakpoints. Interrupts are disabled at hard breakpoints. Interrupts are restored to their former state upon resumption of execution after a hard breakpoint. Soft breakpoints do not affect interrupt state.

Watch Expressions

To obtain the value of a variable, to evaluate an arbitrary expression, or to call a function out of sequence, select 'Add/Delete Watch Expressions' from the INSPECT menu. Enter an expression for evaluation. The result appears in the *watch window.* There are two ways:

1 *Immediate Evaluation.* Enter an expression and click "Evaluate." The expression is evaluated once.

2 *Repeated Evaluation.* Enter an expression and click "Add to top." The expression will be added to the top of the *watch list*. All the entries in the watch list are evaluated every time your program comes to a stopping place.

Being able to evaluate expressions and function calls both periodically and at will is a very powerful facility. You can *change* your program state as well as monitor it. For instance, you might reset the PLCBus, or simulate real-world events by changing the values if inputs and outputs.

Creating Stand-Alone Programs

The object of building a program in Dynamic C is to create stand-alone programs. Thus,

For a controller with EPROM

Once you have burned an EPROM, you may place it in the EPROM socket of your controller. When your controller resets, your program will start running.

For a controller with flash memory

When you compile to a controller with flash memory, Dynamic C places your program code in flash. Therefore, your program is non-volatile. When your controller restarts in run mode, your program will start running.

For a controller with a program in RAM

When you compile to a controller that has EPROM, Dynamic C places your program in RAM. As long as your controller's RAM continues to get power, you can disconnect from Dynamic C, and restart your controller in run mode, and your program will start running.

Help

Dynamic C provides three forms of on-line help.

Standard Windows Help

The first form provides descriptions of the available menus, keystrokes, and dialog box options, as well as other information about using Dynamic C.

Function Lookup

The second form of on-line help provides information about the use Dynamic C library functions. All library functions have descriptive headers which are made known to Dynamic C at startup. When you request help regarding the function, this function header is displayed.

By clicking "Lib Entries," you can *browse* all library functions known to Dynamic C.

Function Assistance

The third form of help is variant of function "lookup." By clicking the 'Insert Call' button, the function assistant will help you place function calls in your program.

The Menus

Dynamic C has eight command menus

FILE EDIT COMPILE RUN
INSPECT OPTIONS WINDOW HELP

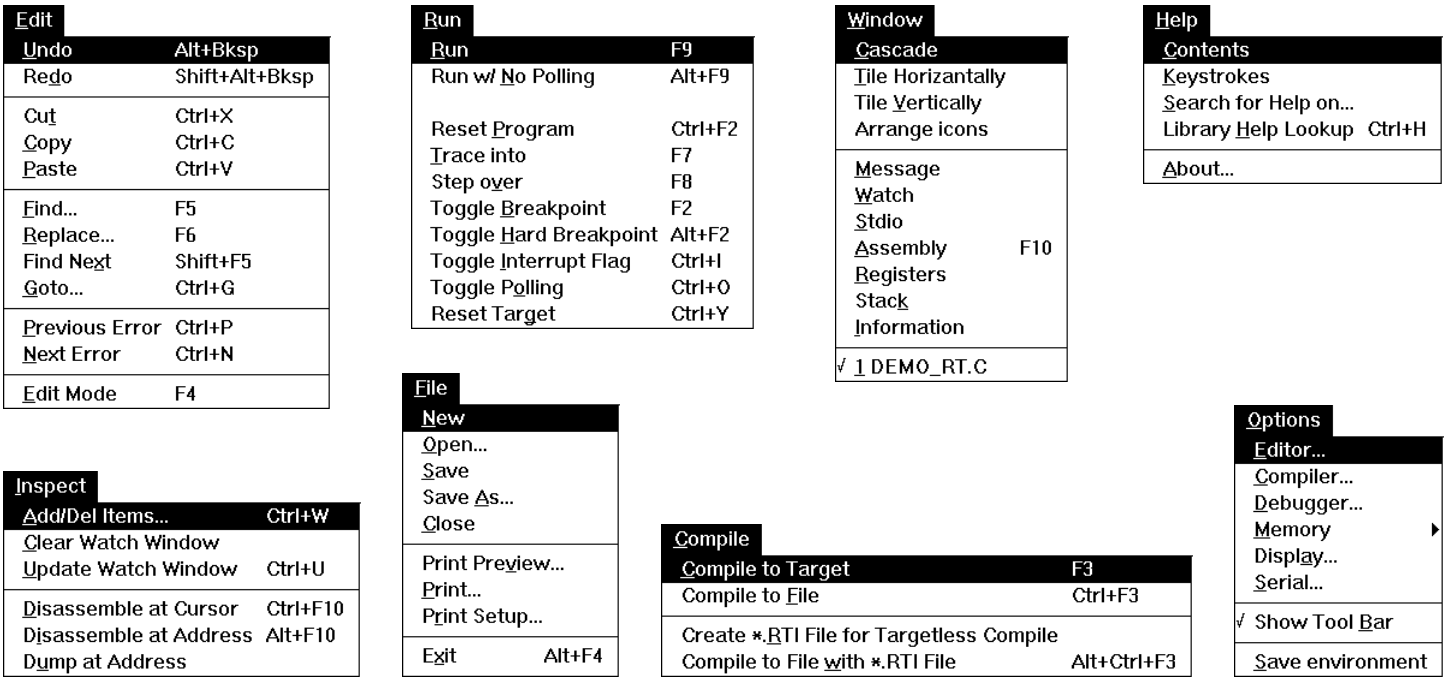
as well as the standard Windows system menus. These are described next.

File Menu

New	Creates a new program in a new window.
Open	Presents a dialog in which to specify the name of a file to open.
Save	Updates the file to reflect your latest changes.
Save As	Allows you to enter a new name for your file and saves it under the new name.
Close	Closes the active window.
Print Preview...	Shows what your printed text will look like.
Print...	Print text from any Dynamic C window.
Print Setup...	Allows you to choose which of your printers to use and to set it up for printing your text.
Exit	Exit Dynamic C.

Edit Menu

Undo	Undoes recent changes in the active edit window. You may undo multiple changes.
Redo	Redoes modifications recently undone.
Cut	Remove selected text from a source file. An image of the text is saved on the "clipboard."



Edit Menu, *continued*

Copy	Make a copy of selected text in a file or in one of the debugging windows. An image of the text is saved on the "clipboard."
Paste	Pastes the text on the "clipboard" (the result of a copy or cut in Dynamic C or some other Windows application) at the current insertion point or in place of selected text.
Find...	Finds specified text.
Replace...	Replaces specified text.
Find Next	Once you have specified search text with the 'Find' or 'Replace' command, the 'Find next' command will find the next occurrence of the same text as you previously specified. If your previous command was Replace, the operation will be a replace.
Goto...	Positions the insertion point at start of the line you specify.
Previous Error	Locates the *previous* compilation error in the source code.
Next Error	Locates the *next* compilation error in the source code.
Edit Mode	Switches Dynamic C back to edit mode.

Compile Menu

Compilation is affected by *compiler options*, *memory options*, and *serial options* (under the OPTIONS menu).

Compile to Target	Compiles your program, loading it in your target controller's memory.
Compile to File	Compiles your program to a file. Dynamic C takes configuration information from your target controller.
Create *.RTI File...	Create a Remote Target Information (RTI) file for your intended controller.
Compile to File with *.RTI File	Compiles your program to a file using an RTI file you created.

Run Menu

Run	Starts program execution from the current breakpoint. Registers are restored, including interrupt status.
Run w/ no polling	This command is identical to the run command with an important exception. When running in polling mode, Dynamic C polls or interrupts the target system every 100 milliseconds.
Stop	Places a hard breakpoint at the point of current program execution.
Reset Program	Resets your program to its initial state.
Trace into	Executes one C statement (or one assembly language instruction) *with* descent into functions.
Step over	Executes one C statement (or one assembly language instruction) *without* descending into functions.
Toggle Breakpoint	Toggles a "soft" breakpoint at the location of the text cursor. Soft breakpoints do not affect the interrupt state; hard breakpoints do.
Toggle Hard Breakpoint	Toggles a "hard" breakpoint at the location of the text cursor. A hard breakpoint disables interrupts when it is reached.
Toggle Interrupt Flag	Toggles interrupt state (enabled or disabled).
Toggle Polling	Toggles polling mode. Normally, Dynamic C polls or interrupts the target controller every 100 milliseconds.
Reset Target	Tells the target system to perform a software reset including system initializations.

Inspect Menu

The INSPECT menu provides commands to manipulate watch expressions, view disassembled code, and produce memory dumps.

Add/Del Watch Expression	Allows you to enter expressions to be evaluated. You can either evaluate this expression immediately by clicking the 'Evaluate' button or you may add it to the watch list by clicking the 'Add to top' button. Expressions in this list are evaluated, with results displayed in the watch window every time the watch window is updated.
Clear Watch Window	Removes all entries from the watch window.
Update Watch Window	Forces the watch expression list to be displayed in the watch window.
Disassemble at Cursor	Disassembles the code at the current editor cursor.
Disassemble at Address	Disassembles the code at the specified address.
Dump at Address	Allows you to look at blocks of raw memory values. You can display values on your screen, or write values to a file.

Options Menu

Editor Options	(1) Change default tab stops (2) Auto-Indent or (3) Remove Trailing Whitespace.
Compiler Options	*Warning Reports.* Tells the compiler whether to report all warnings, no warnings or serious warnings only. *Run-Time Checking.* Check array bounds, invalid pointer assignments, and stack corruption. *Optimize For*. For *size* or for *speed*. *Type Checking.* Perform strict type checking and detect demotion. Generate warnings if pointers to different types are intermixed without type casting. *File Type for "Compile to File."* Se previous remarks. *Object File Option.* Dynamic C can optionally generate an Intel HEX format file.
Debugger Options	(1) Log **printf** statements and other STDIO output to a file, and (2) specify whether to open the STDIO window automatically.
Memory	Specifies memory settings. *Physical.* The size and boundaries of RAM and ROM, or the format of a hex file. *Logical.* Specifies (1) the number of bytes allocated for the run-time stack, (2) the number of bytes allocated to an alternate stack used mainly for stack verification, (3) the heap size, and (4) the size of free space. *Reserve.* Root reserve and XMEM reserve specify how the compiler allocates memory when compiling code whose destination is not specified (that is, *anymem* code).

Options Menu, *continued*

Display	Specifies the appearance of Dynamic C windows, such as foreground and background colors, highlight colors, and font (type face).
Serial	Tells Dynamic C how to communicate with your target controller.
Show Tool Bar	Toggles the tool bar on or off.
Save Environment	Saves your current options settings.

Window Menu

Cascade	Displays your windows neatly overlayed and in order. The window in which you are working is displayed in front of the rest.
Tile Horizontally	When you issue this command, Dynamic C displays your windows in *horizontal* (landscape) orientation.
Tile Vertically	When you issue this command, Dynamic C displays your windows in *vertical* (portrait) orientation:
Arrange Icons	When you have minimized one or more of your Dynamic C windows, they are displayed as icons. This command arranges icons neatly.
Message	Activates or deactivates the message window.
Watch	Activates or deactivates the watch window.
STDIO	Activates or deactivates the STDIO window.
Assembly	Activates or deactivates the assembly window.
Registers	Activates or deactivates the register window.
Stack	Activates or deactivates the stack window.
Information	Activates the information window: The information window tells you how your memory is partitioned and how well your compilation wentand how much space has been allocated to the heap or free space.

Help Menu

Contents	Invokes the on-line help *contents* page.
Keystrokes	Displays information on available keyboard shortcuts and their functions.
Search for Help on	Displays help information for various topics.
Library Help Lookup	Obtains help information for library functions. You may display a list of the library functions currently available to your program. You may then select a function name from the list to receive information about that function. If you click the 'Insert Call' button, the dialog turns into a "function assistant." You will probably need the function help dialog only when you are unfamiliar with or unsure of a function.
About	Displays version number and copyright notice.

The Language

Modules

How does Dynamic C know which functions and global variables in a library to use? A library file contains a group of *modules*. A module has three parts: the *key,* the *header,* and a *body* of code (functions and data).

A module in a library has a structure like this one:

```
/*** BeginHeader func1, var2, .... */
   prototype for func1
   declaration for var2
/*** EndHeader */
definition of func1 var2 and possibly other functions and data
```

The line (a specially-formatted comment)

```
/*** BeginHeader name1, name2, .... */
```

begins the header of a module and contains its *key,* a list of names of functions and data available for reference.

When Dynamic C sees a **#use** directive, it compiles every header, *and just the headers,* in the function library.

Every line of code after the **EndHeader** comment belongs to the *body* of the module until (1) end-of-file or (2) the **BeginHeader** comment of another module. Dynamic C compiles the *entire* body of a module if *any* of the names in the key are used anywhere in your application.

Macros

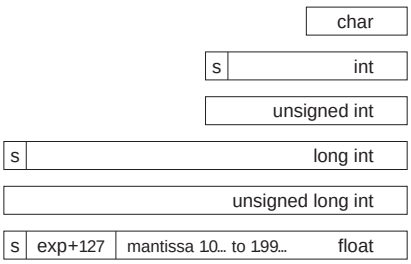
Dynamic C supports ANSI parameterized macro expansion. Dynamic C implements the # and ## macro operators also. Macros are restricted to 32 parameters and 126 nested calls.

Data

Data (variables and constants) have type, size, structure, and storage class. Primitive data types are as follows:

Type	Description
char	8-bit unsigned integer. Range: 0 to 255 (0xFF)
int	16-bit signed integer. Range: –32,768 to +32,767
unsigned int	16-bit unsigned integer. Range: 0 to 65,535
long	32-bit signed integer. Range: –2,147,483,648 to +2,147,483,647
unsigned long	32-bit unsigned integer. Range: 0 to 2^{32} –1
float	32-bit IEEE floating point value. The sign bit is 1 for negative values. The exponent has 8 bits, giving exponents from –127 to +128. The mantissa has 24 bits. Only the 23 least significant bits are stored; the high bit is implicitly 1. (Z180s do not have floating point hardware.) Range: -6.085×10^{38} to $+6.085 \times 10^{38}$

The structures of the primitive data types are shown in relative size in the following figure:



C has string constants and string storage, but not a string data type. There are many functions to manipulate strings.

Storage Classes

Local variable storage can be **static**, **auto**, or **register**. If a variable does not belong to a function, it is called *global*—available anywhere. Global variables are always **static**.

The term **static** means the data occupies a permanent fixed location for the life of the *program*. The term **auto** refers to variables that are placed on the system stack for the life of a *function call*.

The term **register** describes variables that are allocated as if they were static variables, but their values are saved on function entry and restored when the function returns. Thus, **register** variables can be used with reentrant functions as can **auto** variables, yet they have the speed of static variables.

Argument Passing

In C, function arguments are generally passed ***by value.*** That is, arguments passed to a C function are generally copies—on the program stack—of the variables or expressions specified by the caller. Changes made to these copies do not affect the original values in the calling program.

In Dynamic C and most other C compilers, however, arrays are always passed by address. This policy includes strings (which are character arrays).

Dynamic C passes **struct**s by value—on the stack. Passing large **struct** takes a long time and can easily cause your program to run out of memory. Pass pointers to large **struct**s if you are having problems.

If you want a function to modify the original value of a parameter, (1) pass the address of the parameter and (2) design the function to accept the address of the item.

C Language Elements

A Dynamic C application program is a set of files, each of which is a stream of characters which compose statements in the C language. The basic elements of the C language include:

keywords	words used as instructions to Dynamic C
names	words used to name your functions and data
numbers	literal numeric values
strings	literal character values enclosed in quotes
operators	symbols used to perform arithmetic
punctuation	symbols used to mark beginnings and endings
directives	words starting with # that control compilation

The following material presents Dynamic C's few differences from standard C.

Keywords

The following Dynamic C keywords are not in ANSI C:

abort	Jump out of, and terminate, a costatement.
anymem	Allow the compiler to determine in which part of memory a function will be placed.
costate	Indicates the beginning of a costatement.
debug	A function is to be compiled in debug mode.
firsttime	A function is to be a **waitfor** delay function.
interrupt	A function is an interrupt service routine.
nodebug	A function is not compiled in debug mode.
norst	A function does not use **RST** for breakpoints.
nouseix	A function uses SP as a stack frame pointer.
NULL	The null pointer. (This is actually a macro.)
pop	A keyword used in conjunction with directives **#memmap** and **#class**.
protected	Declares a variable to be "protected" against system failure.
push	A keyword used in conjunction with directives **#memmap** and **#class**.
ret	Indicates that an interrupt service routine written in C uses the **ret** instruction.
reti	Indicates that an interrupt service routine written in C uses the **reti** instruction.
retn	Indicates that an interrupt service routine written in C uses the **retn** instruction.
root	Indicates a function is to be placed in root memory.
segchain	Identify the beginning of a function chain segment (within a function).
shared	Indicates that changes to a multi-byte variable (such as a **float**) are atomic.
size	Declares a function to be optimized for size.
speed	Declares a function to be optimized for speed.
subfunc	Begins the definition of a subfunction.
useix	A function uses IX as a stack frame pointer.
waitfor	Waits for a condition, completion of an event, or a delay, in a costatement.
xdata	Declares a block of data in extended memory.
xmem	Indicates that a function is to be placed in extended memory.
xmemok	Indicates that assembly code embedded in a C function can be compiled to extended memory.
xstring	Declares strings in extended memory.
yield	Pauses a costatement temporarily, allowing other costatements to execute.

Names

Names are distinct up to 16 chars, but may be longer.

Strings

Dynamic C treats strings the same way standard C does. Character constants, however, are limited to a single byte.

Directives

Dynamic C has several directives not found in ANSI C:

- **#asm [** *options...* **]**
  #endasm

  Begin and end blocks of assembly code.
- **#class [push] [** *options...* **]**
  #class pop

  Control the default storage class for local variables.
- **#debug**
  #nodebug

  Enable or disable debug-code compilation.

- **#fatal "..."**
  #error "..."
  #warns "..."
  #warnt "..."

  Instructs the compiler to act as if a fatal error (**#fatal**), an error (**#error**), a serious warning (**#warns**) or a trivial warning (**#warnt**) were issued.
- **#funcchain** *chain_name name*

  Add a function, or another function chain, to a function chain.
- **#interleave**
  #nointerleave

  Controls whether Dynamic C will intersperse library functions with your program's functions during compilation. **#nointerleave** forces your functions to be compiled first.
- **#int_vec** *[const] function*

  Loads the address of *function* in the interrupt vector table at an offset equal to *const*.
- **#KILL** *name*

  If you wish to redefine a symbol found in the BIOS of your controller, you must first "kill" the prior name.
- **#makechain** *chain_name*

  Create a function chain.
- **#memmap [push] [** *options...* **]**
  #memmap pop

  Control the default memory area for functions.
- **#use** *pathname*

  Activates a library (named in LIB.DIR) so modules in the library can be linked with your application program.
- **#useix**
  #nouseix

  Control whether functions use the IX register as a stack frame reference pointer or the SP (stack pointer) register.

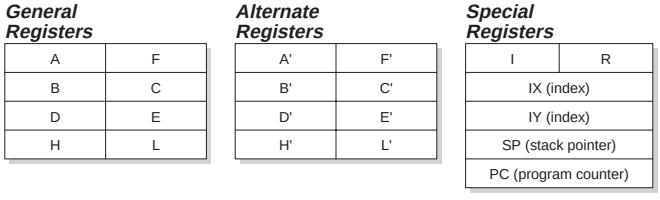
Using Assembly Language

To place assembly code in your program, use the **#asm** and **#endasm** directives.

You can place a C statement within assembly code by placing a 'C' in column 1. You can use C variable names in Dynamic C assembly language.

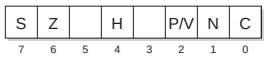
Register Summary

The Z180 has the following basic register set.



Register A is the accumulator. Registers B–L are general purpose registers and can be coupled in pairs BC, DE, HL for 16-bit values. Registers B, C, D, and E may also be coupled (and called BCDE) for 32-bit values.

Register F (flags) holds status bits:



S: sign bit
Z: zero bit
H: half-carry
P/V: parity or overflow
N: negative op
C: carry

The alternate set of registers (A'–L') is often used to save and restore register values. There are instructions to swap register sets.

The PC is the program counter; SP is the stack pointer. The IX and IY registers are index registers. The I register is the interrupt vector register. (You can ignore the R register.)

Dynamic C uses the HL register pair (1) to pass the first 16-bit argument, and (2) to return a 16-bit function result. Dynamic C uses the BCDE register group (1) to pass the first 32-bit argument and (2) to return a 32-bit function result.

The Z180 has many other special-purpose registers. So also do the Zilog PIO, SCC, and KIO chips.

Interrupt Service Routines

Dynamic C allows you to write interrupt service routines in C, although assembly routines tend to be more efficient than the C equivalent functions.

In general, an interrupt routine written in C saves and restores all registers. It disables interrupts on entry and reenables interrupts on exit. Interrupt routines cannot have parameters and must be void (they cannot have "function results").

Interrrupt Vectors

Interrupt vectors are of two types. The first type handles modes 0, 1 and non-maskable interrupts. This type includes:

```
08h:   jp restart_service        ; mode 0 int
38h:   jp interrupt0_service     ; mode 1 int
66h:   jp nmi_service            ; non-maskable
```

The second type handles the mode 2 interrupt used by Z180 peripheral devices, Z180 internal I/O devices and Dynamic C. This involves a 256-byte table, identified by the I register, that can contain addresses of up to 128 interrupt service routines.

To set interrupt vectors in the page specified by the I register, use the following preprocessor directive:

```
#INT_VEC ( const_expr ) function_name
```

The constant expression is the offset, in bytes, of the interrupt vector, which is always an even number from 0 to 126. The function name is the name of the interrupt routine.

Dynamic C common interrupt vectors are given in the table following. Some Z-World controllers use additional vectors.

Standard Vectors

Addr	Name	Description
0x00	INT1_VEC	INT1, expansion bus attention.
0x02	INT2_VEC	INT2 vector.
0x04	PRT0_VEC	Programmable timer channel 0
0x06	PRT1_VEC	Programmable timer channel 1
0x08	DMA0_VEC	DMA channel 0
0x0A	DMA1_VEC	DMA channel 1
0x0C	CSIO_VEC	Clocked serial I/O
0x0E	SER0_VEC	Asynchronous Serial Channel 0
0x10	SER1_VEC	Asynchronous Serial Channel 1

Interrupt Priorities, Highest to Lowest

Trap (Illegal Instruction)

NMI (non maskable interrupt)

INT 0 (Maskable interrupts, level 0. Three modes)

INT 1 (Maskable interrupts, level 1. PLCBus attention line)

INT 2 (Maskable interrupts, level 2)

PRT channel 0

PRT channel 1

DMA channel 0

DMA channel 1

Clocked serial I/O

Serial Port 0

Serial Port 1

Costatements

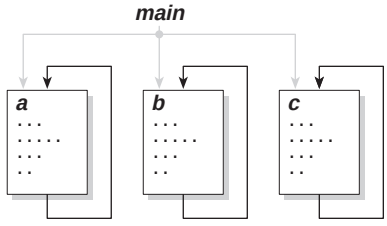
Dynamic C supports multi-threaded real-time programming. You may use one of the real-time kernels or you may use ***costatements***. Costatements give you *cooperative multi-tasking* within your application.

The advantages are several:

- Costatements are a feature built into the language.
- Costatements are cooperative instead of preemptive.
- Costatements operate within a single program.

Costatements are blocks of code that can suspend their own execution at various times for various reasons, allowing other costatements or other program code to execute. Costatements operate concurrently. For example, the code shown here will operate as shown in the diagram: 3 processes, **a**, **b**, and **c**, operate independently and concurrently.

```
main(){
  int x, y, z;
  ...
  for(;;){
    costate a {
    ...
    }
    costate b {
    ...
    }
    costate c {
    ...
    }
  }
}
```



It is only when you have more than one task that they can be considered *cooperative*, because it is only when you have more than one task that any task can execute in the idle time of another task. Nevertheless, some single tasks are easier to write using costatements. Costatements can be used, for instance, to create delays.

A typical set of costatements will execute in an endless loop. This is not a requirement, however.

Costatements are *cooperative* because they can suspend their own operation. There are three ways they do this.

1 Wait for an event, or a condition, or the passage of a certain amount of time. For this, there is the **waitfor** statement.

2 Use a **yield** statement to yield temporarily to other costatements.

3 Use an **abort** statement to cancel their own operation.

Since costatements can suspend their own execution, they can also resume their own execution. Placing costatements in a loop is the simplest way for you to give each costatement a chance to progress in its turn.

Costatements can be active (ON) or inactive (OFF). You may declare a costatement "always on," "initially on," or "initially off. A costatement that is *initially on* will execute once and then become inactive. A costatement that is *initially off* will not execute until it is started by some other part of your program. Then it will execute once and become inactive again.

Remote Download

Z-World provides field programmability for its controllers. You can create a downloadable program file by selecting the appropriate compiler option. The Z-World Download Manager (DLM), resident in a controller, will receive your program, place it in memory, and start it running. Remote downloading requires a communications program such as ProComm that has XMODEM transfer protocol available.

Run-Time Error Processing

Dynamic C's standard error handler prints error messages to the STDIO window. The standard error handler will not work when your software is running stand-alone. You should provide your own error handler:

```
void my_handler( uint code, uint address ){
   my error processing code...
}
main(){
   ...
   ERROR_EXIT = my_handler;
   some statements...
   (*ERROR_EXIT)( code, addr );
   some statements...
}
```

A built-in Dynamic C symbol—ROM—is set to 1 if you are compiling to an EPROM file. Use this variable to install your own error handler conditionally:

```
#if ROM
   ERROR_EXIT = my_handler;
#endif
```

Long Jumps

You can perform error recovery using Dynamic C's **setjmp** and **longjmp** functions. If you detect an error anywhere in your

program, you can make a "long jump" to a safe location and perform necessary recovery tasks. Usage typically involves jumping from a deeply nested function back to your main program.

The **setjmp** function marks a place in your code and saves the stack pointer and important registers. The **longjmp** function causes a return to the place marked by **setjmp**. The processor stack is immediately "unwound" and a known state is restored.

This is how you do a long jump:

```
// probably in main()
   jmp_buf savreg;   // make a save buffer
   ...
   if( setjmp(savreg) ){
      code to recover from the error
   }
   ...
// then, somewhere, deeper in your code...
   if( big error ) longjmp(savreg,1);
```

When **longjmp** executes, execution resumes immediately after the call to **setjmp** and the value returned by the call to **setjmp** is the same as the second argument passed to **longjmp**. This value can be your error code as long as it is non-zero (**setjmp** returns 0 when you call it directly.)

Watchdog Timer

Most Z-World controllers have a watchdog timer. Briefly stated, a watchdog timer will reset your system after a certain period (about 1.5 seconds, typically) if your software does not reset the watchdog timer within that period. This safety feature ensure that your program is functioning.

Protected Variables

Your program may need to recover protected variables at restart. However, if your program has never run before, your program must initialize protected variables.

The function **_prot_recover** recovers protected variables; the function **_prot_init** initializes them. The function **_sysIsSuperReset** calls whichever of these functions is appropriate at startup (if you call **_sysIsSuperReset**).

Efficiency

There are a number of methods you can use to reduce the size of your program, or to increase its speed.

Nodebug Keyword

Dynamic C places an **RST 28H** instruction in debug code at the beginning of each C statement to provide locations for breakpoints. These "jumps" to the debugger consume one byte and about 25 clocks of execution time for each statement. Your function will not have **RST 28H** instructions if you use the **nodebug** keyword in the function declaration:

```
nodebug int myfunc( int x, int z ){
   ...
}
```

Static Variables

Static variables are much more efficient on the Z180 than **auto** variables. In Dynamic C, the default local storage class is **static**, while most C compilers use **auto**. Use **auto** variables in reentrant or recursive functions.

Execution Speed

Under Compiler Options, you can set a switch to optimize for speed or for size. The default is size.

Subfunctions

Subfunctions, extensions in Dynamic C, allow often-used code sequences to be turned into a "subroutine" within the scope of a C function.

```
func(){
   int aname();
   subfunc aname: { k = inport (x); k + 4; }
   ...
   ... aname(); ...
   ...
   ... aname(); ...
...
}
```

The subfunction is prototyped as if it were a regular function. It must be **static** and may not have any arguments. Variables used within the subfunction must be available within the scope of the parent C function. The actual code after the **subfunc** keyword can appear anywhere in the enclosing function. You indicate the return value, if any, by placing an expression followed by a semicolon at the end of the subfunction body. This causes the expression value to be loaded into the primary register (HL or BCDE).

All subfunction calls take three bytes, low overhead compared to some simple expressions. For example, the expression ***ptr++** can generate 14 bytes or more.

Substituting the following code:

```
static char nextbyte();
subfunc nextbyte: *ptr++;

nextbyte();
...
nextbyte();
...
```

can save ten or more bytes for each occurrence of **nextbyte**. Subfunctions can also make a program easier to read and understand, if you use descriptive names for obscure expressions. The advantage of the subfunction over a regular function is that it has access to all the variables within the program and the calling overhead is low.

Subfunction calls cannot be nested.

New Features

Revision 5.0 of Dynamic C incorporated these features:

- Macros with Parameters
- Function Chaining (specifically including a **_GLOBAL_INIT** function chain).
- Printing the contents of any window.
- C operators in constant expressions in assembly code.
- A hexadecimal memory dump command.
- Horizontal and vertical tiling options.

- Toolbar.
- New COMPILE menu options. Specifically, targetless compilation, and creation of downloadable code.
- Function "Assistant"
- Changes to the **Codata** structure
- New reset and initialization functions

Backward Compatibility

The **#GLOBAL_INIT** directive still works as it did in prior releases of Dynamic C.

You can still "download to RAM." It's a compiler option. However, this capability is no longer being supported.

Software Libraries

These are the libraries included with Dynamic C. This list is subject to change, as new products are introduced, and software gets revised.

5KEY.LIB	The basic "five-key" operator interface for the PK2100 and PK2200 series controllers.
5KEYEXTD.LIB	Extensions to the "five-key system."
96IO.LIB	Driver for the BL100's DGL96 daughter board.
AASC.LIB	Abstract Asynchronous Serial Communication.
AASCDIO.LIB	STDIO-specific support for the AASC library.
AASCSCC.LIB	SCC-specific support for the AASC library. (The SCC is the Zilog 85C30 Serial Communication Controller.)
AASCUART.LIB	XP8700 support for the AASC library.
AASCZ0.LIB	Z0-specific support for the AASC library. Z0 is the Z180 serial port 0.
AASCZ1.LIB	Z1-specific support for the AASC library. Z1 is the Z180 serial port 1.
AASCZN.LIB	ZNet-specific support for the AASC library.
BIOS.LIB	Contains prototypes of functions and declarations of variables defined in, and used by, the BIOS.
BL11XX.LIB	Functions for the BL1100.
BL13XX.LIB	Functions for the BL1300.
BL14_15.LIB	Functions for the BL1400 and BL1500 series.
BL16XX.LIB	Functions for the BL1600.
CIRCBUF.LIB	Circular buffers functions (used by the AASC).
CM71_72.LIB	Functions for the CM7100 and CM7200 series.
COM232.LIB	Serial communication functions for the COM ports on the Z104.
CPLC.LIB	Functions for PK2100, PK2200, and BL1600.
DC.HH	Definitions basic to Dynamic C.
DEFAULT.H	Contains lists of **#use** directives for various Z-World controllers. Dynamic C automatically selects the list appropriate for your controller.
DMA.LIB	Support functions for the Z180 on-chip DMA channels.
DRIVERS.LIB	Drivers for some hardware devices.
EZIO.LIB	Driver for a board-independent unified I/O space.
EZIOCMMN.LIB	Common definitions for all EZIO libraries.
EZIOPBDV.LIB	PLCBus support for the EZIO library.
EZIOPK23.LIB	PK2300 function support for the EZIO library.
EZIOPLC.LIB	PLCBus functions for boards that have native PLCBus ports (BL1200, BL1600, PK2100, and PK2200).
FK.LIB	New "five-key" operator interface for the PK2100 and PK2200.
IOEXPAND.LIB	Driver functions for BL1100 series daughter boards.
KDM.LIB	Driver for keyboard/display modules.
LCD2L.LIB	LCD support for the PK2100 and PK2200.
MATH.LIB	Useful mathematical functions.
MISC.LIB	Miscellaneous KDM support.
MODEM232.LIB	Modem functions for the PK2100 and PK2200.
NETWORK.LIB	Opto22 9-bit binary protocol to support master-slave networking.
PBUS_LG.LIB	PLCBus support for the BL1100.
PBUS_TG.LIB	PLCBus support for the Tiny Giant.
PK21XX.LIB	PK2100 functions.
PK22XX.LIB	PK2200 functions.
PLC_EXP.LIB	PLCBus functions for boards that have native PLCBus ports (BL1200, BL1600, PK2100, and PK2200).
PRPORT.LIB	Parallel port communication protocol between a controller and a PC.
PWM.LIB	Pulse width modulation functions.
RTK.LIB	Real-time kernel.
S0232.LIB	Serial communication driver for SIO port 0 on the BL1100.
S1232.LIB	Serial communication driver for SIO port 1 on the BL1100.
SCC232.LIB	Serial communication driver for the ports on Zilog's Serial Communication Controller.
SERIAL.LIB	Serial communication functions for ports 0 and 1 of the Z180 and the SIO. *Obsolete*.
SRTK.LIB	Simplified real-time kernel.
STDIO.LIB	Functions relating to the STDIO window.
STRING.LIB	(ANSI) functions for manipulating strings.
SYS.LIB	General system functions.
TGIANT.LIB	Functions for the Tiny Giant.
VDRIVER.LIB	Virtual driver functions.
XMEM.LIB	Function support for extended memory.
XP82XX.LIB	Driver for the XP8200.
XP87XX.LIB	Driver for the XP8700.
XP87XX2.LIB	Driver for a second XP8700.
XP88XX.LIB	Driver for the XP8800.
Z0232.LIB	Serial communication driver for Z0. Z0 is the Z180 serial port 0.
Z104.LIB	Functions for the Z104 (no longer available).
Z1232.LIB	Serial communication driver for the Z180 serial port 1.
ZNPAKFMT.LIB	Lower level functions supporting the ZNet.

//

Dynamic C must be installed on a hard disk and requires about 4M of disk space. You must be running in 386 enhanced mode, using Windows 3.1, Windows 95, or Windows NT, on a machine having a 386SX processor or better. You must have at least 4M RAM to run Dynamic C and one free serial port for communicating with your target controller.